ACM ICPC World Finals 2012

Solution sketches

Disclaimer *This is an unofficial analysis of some possible ways to solve the problems of the ACM ICPC World Finals 2012. Any error in this text is my error. Should you find such an error, I would be happy to hear about it at* `austrin@kth.se`.

Also, note that these sketches are just that—sketches. They are not intended to give a complete solution, but rather to outline some approach that can be used to solve the problem. If some of the terminology or algorithms mentioned below are not familiar to you, your favorite search engine should be able to help.

Finally, I want to stress that while I'm the one who has written this document, I do not take credit for the ideas behind these solutions—they come from many different people.

— Per Austrin

Summary

My guesstimated order for the problems to be solved was BKECLGDAFIHJ. The actual order in which the problems were solved was BDKLCEGIFA, with HJ left unsolved, which means I was less off than usual though I certainly overestimated problem D and underestimated problem E. In terms of number of teams that ended up solving each problem, the numbers were:

Problem	A	B	C	D	E	F	G	H	I	J	K	L
Solved	2	110	76	96	31	2	8	0	11	0	67	16
Submissions	51	202	251	278	300	29	45	12	113	0	289	90

In total there were 419 Accepted submissions, 924 Wrong Answer submissions, 243 Time Limit Exceeded submissions and 74 Run Time Errors. The most popular language was C++ by a wide margin: 1459 submissions compared to 220 for Java. There was also one single submission made in C (which was accepted, on problem B).

Congratulations to St. Petersburg State University of IT, Mechanics and Optics, the 2012 ICPC World Champions!

Problem A: Asteroid Rangers

This problem is a variant of the so-called *kinetic spanning tree* problem (though this knowledge is pretty much irrelevant for solving the problem). The main observation is that the only time the minimum spanning tree could possibly change is when the distances $d(i_1, j_1)$ and $d(i_2, j_2)$ between two pairs of asteroids become equal. Finding these times for two given pairs (i_1, j_1) and (i_2, j_2) just amounts to solving a quadratic equation. Thus, there are at most $2\binom{\binom{n}{2}}{2} \leq 1\,499\,400$ possible times when the spanning tree switches.

We go through these times in increasing order of time. Consider a time t at which two pairs (i_1, j_1), (i_2, j_2) become equidistant. If exactly one of the two pairs, say (i_1, j_1) is an edge in the current spanning tree, there is a possibility that it is switched for (i_2, j_2) at time t (this happens exactly if (i_1, j_1) is on the path from i_2 to j_2 in the current tree, *and* $d(i_2, j_2)$ becomes

smaller than $d(i_1, j_1)$ after time t). The easiest way to check this is to simply recompute the spanning tree at time $t + 0.5 \cdot 10^{-6}$ and see if it changed (the problem statement guarantees that the MST can not change more than once in the interval between t and $t + 0.5 \cdot 10^{-6}$).

In principle this gives an $O(n^6)$ solution (if using a quadratic MST implementation), but it should be pretty clear that the check for whether one of the pairs is in the tree cuts down actual runtime bound significantly—heuristically one would only have to consider at most $O(n^3)$ of the $O(n^4)$ candidate switch times. Incidentally, it seems very hard to create cases where the answer is more than a few hundred (at least we didn't figure out how to do it).

Problem B: Curvy Little Bottles

This was one of the easiest problems. The volume up to point x is simply

$$V(x) = \int_{t=x_{\text{low}}}^{x} P(x)\mathrm{d}x = \sum_{i=0}^{n} \frac{a_i}{i+1}(x^{i+1} - x_{\text{low}}^{i+1}),$$

and to find the value of x for which $V(x)$ is some multiple of inc one can just binary search.

Problem C: Bus Tour

This is a variant of the TSP problem, with some non-standard constraints on the tour. For standard TSP there is a well-known $O(n^2 2^n)$ solution using dynamic programming. It turns out that we can achieve the same time complexity for this problem but we need to do a little more work.

Let s be either the headquarters or the attraction, X be a subset of hotels, and $i \in X$ be some hotel. Similarly to the usual DP solution, define $C(X, s, i)$ to be the minimum cost of a path that starts in s, goes through all the nodes in X, and ends in i. All such values can be computed using dynamic programming in time $O(n^2 2^n)$.

Next, go over all subsets X of size $\lfloor h/2 \rfloor$ and compute the length of the minimum tour which uses this specific subset as the first $\lfloor h/2 \rfloor$ hotels. Given X, the length of the tour can be computed by composing four paths of the form above (this uses that the graph is undirected; if the graph was directed one would have to compute the length of paths that also *end* at the headquarters/attraction).

Problem D: Fibonacci Words

There's a relatively straightforward dynamic programming solution which is probably the first one typically comes up with, but here's a much cooler solution found by Derek Kisman: we can think of the process of going from $F(n-1)$ to $F(n)$ as a string replacement rule: every 1 becomes replaced by 10, and every 0 becomes replaced by a 1. In order to compute the number of occurrences of p in $F(n)$, we can apply these rules backwards to obtain a shorter string p', and the task is now to find the number of occurrences of p' in $F(n-1)$ (if p ended in a 1 there are two possibilities for p' since we don't know whether that last 1 came from a 1 or a 0 in $F(n-1)$). Recursing this process, we are ultimately looking for the number of occurences of either 1 or 0 in $F(n')$ for some $n' < n$, which simply equals $f(n-1)$ and $f(n-2)$, respectively.

Problem E: Infiltration

In graph-theoretic terms, the problem asks for a minimum dominating set in a tournament. The solution is essentially just exhaustive search, and the main difficulty is to realize that it works. It is not hard to prove that a simple greedy algorithm always produces a dominating set of size at most 6. Therefore, we can start by computing the greedy solution and then trying all smaller sets of vertices (there are at most $\sum_{i=1}^{5} \binom{75}{i} = 18\,545\,215$ such sets). To make it fast enough, one should use bitmasks for quickly checking whether a given set is a solution.

In fact, the largest possible answer is actually only 5, so the greedy step is not needed. But proving this is not easy and knowing it is of course not necessary to solve the problem.

Problem F: Keys

Despite its relatively straightforward statement, this was a somewhat nasty problem with several special cases. Let us start with key cost. Whenever there are keys of both type on the same ring, one of the key types has to be "evacuated". Typically, we want to evacuate the key that is the least frequent on the ring (in case both are equally frequent we will try both possibilities; this happens for at most 13 rings). However, if this results in no free rings for the evacuated keys we have to reserve some ring for the evacuated keys (again we can try all possibilities). Another special case, illustrated by the sample data, is that if there is only one ring and both types of keys then the problem is impossible.

Once the key conflicts have been resolved, we can compute the ring costs by standard dynamic programming.

Problem G: Minimum Cost Flow

The first step is to figure out where to put the water pressure level. Or rather, to blindly try all possibilities for the water pressure, which incurs a factor of N on the running time.

Given the water level, consider the induced subgraph on the set of points that are below water. This consists of a set of connected components $C_1, \ldots, C_r$, which we can think of as forming a (complete, weighted) graph G, in which the distance from component C_1 to component C_2 is the minimum distance between a point in C_1 and a point in C_2 which both have an available pipe hole. We want to find a path from the component containing s (say, C_1), to the component containing t (say, C_r), possibly using some intermediate components at shortcuts. The total cost of the path is the sum of distances of edges used, plus the cost of plugging the holes in the used components. The cost of plugging the holes in some intermediate component C_i is $0.5 \cdot (\#\{\text{open holes in } C_i\} - 2)$, because all holes except the two that we use to connect C_i to the previous and next component need to be plugged. Similar calculations apply for the components C_1 and C_r. Note that intermediate components must have at least two holes (and the source and destination components must have at least one hole). A special case has to be dealt with: when s and t are in the same component, no path is needed and the plugging cost is just $0.5 \cdot \#\{\text{holes in } C_1\}$.

Problem H: Room Service

This problem can be solved with dynamic programming and some geometry. First, it is intuitively clear (but a bit tricky to prove) that the edges should be visited in order, and we can guess which of the N edges we visit first.

Suppose for a while that the cleaning robot's path does not visit any of the corners of the polygon. Then, every time the robot "bounces" against an edge of the polygon, the incoming angle of the robot's path should equal the outgoing angle. With this observation, one can, given a starting point and a destination point and a sequence of edges to bounce against, compute the path (and in particular the length) the robot should take by reflecting the destination point around the bounce segments in reverse order. After these reflections, the path of the robot is "unwinded" to the straight line from the starting point to the new position of the destination.

However, when we are at a corner, the notion of the incoming angle breaks down and we can actually go off in any direction. This suggests a dynamic program where the state is what vertex we are currently at (either the robot's starting point or one of the polygon vertices). Given that we are visiting the edges in order and we know which edge we visit first, the current vertex completely determines which edges are left to visit. To compute the answer from this position, we try all possibilities for the next vertex to visit, and compute the cost of bouncing all the way there as described in the previous paragraph (but note that it might not be possible from the current vertex to some subsequent vertex because it would result in the path going outside the polygon). This gives an $O(N^3)$ solution.

Problem I: A Safe Bet

First, we trace the route of the laser beam from the entry through the maze. This can be done in $O(N \log N)$ time (where we let $N = n + m$ be the total number of mirrors) by keeping track of dictionary of the mirror positions in each row and column. This trace can be represented as a set of horizontal and vertical line segments.

If the laser beam exits at the exit, we're done. Otherwise, we run the same trace backwards from the exit position, giving a second set of horizontal and vertical line segments.

After this, the set of positions where a mirror can be inserted are simply all intersection points between line segments from the two traces. This can be found in $O(N \log N)$ time using a standard sweepline approach, though one needs a datastructure that can do quick range counting, e.g., a Fenwick tree.

Problem J: Shortest Flight Path

We construct the following graph: the vertices are all the airports, together with all intersection points between the "safety circles" around the airports (i.e., all points that are at distance R from two airports). There are at most $2\binom{N}{2} \leq 600$ such intersection points. There is an edge between two points if the great circle arc between the two points lies completely inside the allowed flight region (i.e., the union of the "safety circles").

The main difficulty of the problem lies in constructing this graph, after that it is fairly straightforward graph problem which we leave to the reader. First, we need to find the intersection points, i.e., compute the intersections of two circles in 3D. which is a bit messy. Second, we need to check if the arc between two points lies inside the allowed flight region. This can be done by again using circle-circle-intersections to compute which part (if any)

of the arc is covered by each airport, and then computing the union of all these parts. This gives an $O(N^5 \log N)$ algorithm (though since N is so small, using a quadratic interval union which gives an $O(N^6)$ solution works fine).

Problem K: Stacking Plates

If all plates had different sizes, the problem would be completely trivial: take all the plates, sort them in order (but keep track of which stack they came from). Then, the number of splits we have to do is the number of positions where we switch between stacks (in the sorted list of plates). The number of join operations is always #splits $+$ #*stacks* -1, so finding the number of splits is enough.

If there are ties, we get groups of plates that are of the same size, and we have to figure out in which order to put them so as to minimize the number of positions where we switch between stacks. This can be done with dynamic programming.

Problem L: Takeover Wars

The main observations that need to be made are the following:

1. If doing a takeover, we should always take over the largest company of the opponent (in particular, if we can not take over that company we should not do a takeover).

2. If doing a merge, we should always merge the two largest companies that we have.

While intuitively obvious, proving them is a bit tedious, so have fun with that.

With these observations, one can try a simple recursive tree search. This actually works (as long as you always try doing a takeover, when possible, before doing a merge), except that you will probably run out of stack size and crash.

Why should it work? Well, here are two more observations (easy to prove given the first two):

3 If a takeover is possible at any time after a merge move has been made, then that takeover will be winning.

4 After a takover, the opponent will be forced to merge.

Taken together, this implies that it is only the first move of the game in which there is actually a choice. Thus, one can just try the (at most) two possibilities for that move and then easily simulate the rest of the game.